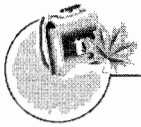

第8章

运行用户程序

经过了不懈的努力，现在我们的操作系统已然有了一点大家风范。

然而，无论我们如何强调操作系统的重要性，如何想尽办法来提高操作系统的运行效率和稳定性，都不得不承认一个事实，那就是我们的操作系统目前似乎什么有用的事都做不了。它不能用来听歌、看电影，不能玩游戏、上网……

事实上，一个操作系统的作用不是发挥在功能上，而是发挥在机制上。就像 Windows 一样，仅仅装上了 Windows 操作系统，基本上是用不了的，你要额外装一个 QQ 工具才能聊天，要装一个客户端，才可以玩网络游戏，所以，操作系统并不能让你直接去做什么，而是在你想做什么的时候，尽最大可能地为你提供足够的支持，当然操作系统的作用还包括了资源的适当分配和系统监控，保证了玩游戏的时候不会因为程序抢占了所有的资源而导致你无法聊天。

与之对应的，应用程序负责的正是功能上的实现。在强大的操作系统支持下，应用程序能够更自由地实现各种复杂功能。这些程序虽然可以通过与操作系统相类似的方法编写和编译并被操作系统所调用，但是很多时候，它们不能享受与操作系统相同的待遇，并且要受到来自于操作系统的限制和管理。

从这个角度上看，操作系统就像一位司令官，自己并不会亲自上战场打仗，而是分配和调动手下的士兵，尽可能为他们提供充足的枪支弹药去赢得战场上的胜利。而这些冲锋陷阵的沙场战士们就是用户应用程序。

我们的操作系统现在已经成长为了一名合格的司令官。于是，本章的重点内容将会是如何让我们的操作系统有效地运行应用程序。

8.1 二进制程序的运行方法

其实，让我们的操作系统运行一个应用程序是非常简单的。不要忘了，应用程序也是程序，操作系统也是程序，让一个程序去调用另一个程序，又会难到哪儿去呢？

在 ARM 体系结构中，实现程序的跳转有很多种方法，这些内容在前面我们也都有介绍。

但有一点我们需要强调，我们这里的应用程序指的都是独立程序。因为有的嵌入式操作系统将应用程序定义成实现具体功能的程序，在这样的操作系统中，应用程序往往会与操作系统内核统一编程以实现程序的简化。

既然程序是独立的，那么一些简单的程序跳转方法就不能实现应用程序的运行了。此时我们可以获取应用程序的入口地址，使用 mov 指令来实现。

应用程序的入口地址应该怎样获取呢？目前没有什么好办法，只能让操作系统事先跟应用程序商量好，比如让一个应用程序在编译时就链接到某地址处，然后操作系统要把应用程序复制到链接时的内存位置，最后调用 mov 指令将 PC 寄存器的值赋值成该地址，实现程序运行。

下面我们通过一个小小的例子，看看最简单的应用程序调用过程是如何实现的。

代码 8-1

```
int main( ){
    const char *p="this is a test application\n";
    while(*p){
        *(volatile unsigned int *)0xd0000020=*p++;
    };
}
```

代码 8-1 就是这样一个应用程序。为了保证代码的简单，程序仅仅打印出 " this is a test application\n " 这个字符串，以证明应用程序的正常运行。

然后，我们需要将代码 8-1 保存到文件之中，命名为“main.c”，把这个文件保存到代码根目录的“tools”文件夹里，然后切换到这个文件夹并使用如下命令编译这段程序。

命令 8-1

```
arm-elf-gcc -e main -nostartfiles -nostdlib -Ttext 0x30100000 -o
main main.c
```

这些编译选项的具体含义前面都已经介绍过了。虽然这里编译的是用户应用程序，但我们却不能像通常编写程序那样使用标准库函数和启动程序。广义上讲，标准库函数也属于应用程序的范畴，但此时这些函数库尚未建立。因此，目前的应用程序会显得非常简陋，我们既不能在程序中使用标准库中的函数和头文件，也不能在编译程序的过程中链接它们。

与此同时，为了保证程序能够正常运行，我们必须在编译时指定程序的运行地址，让它恰好落在有效的内存中。于是程序选定了 0x30100000 这个地址。回忆一下前面的内容，我们在 MMU 一节中将实际的物理地址映射到了虚拟地址中相应的位置。因此在虚拟地址空间中，0x30000000 之后 8M 的内容都是有效的，留出操作系统自身所占的内存之后，0x30100000 这个位置就非常合适了。

既然代码已经编译完成了，那么我们的应用程序是不是就可以运行了？

不，现在还不是时候。要知道，GCC 默认只会生成 ELF 格式的文件，这是一种非常常用的可执行程序的文件格式，但因为这样一个可执行的文件包含了除代码和数据之外的附加信息，所以不能直接拿来运行。有关 ELF 文件格式的详细内容，我们稍后会有介绍。

这样一来，想要运行这段应用程序，必须首先将由 GCC 编译生成的文件转换成二进制的形式，也就是仅由代码和数据所组成的文件，使用的命令如下：

命令 8-2

```
arm-elf-objcopy -O binary main main.bin
```

此时我们需要将“main.bin”文件复制到“filesystem”文件夹中：

命令 8-3

```
cp main.bin filesystem
```

不要忘了，现在在我们的操作系统中已经能够支持文件系统了。也就是说，程序完全可以以文件的形式来读取数据，不再需要使用存储器的原始操作方法了。

于是我们便可以使用上一章提到的方法，将 filesystem 文件夹中的内容制作成 romfs 格式的映像文件，并将这个映像文件复制到源代码根目录的“ram.img”文件中。

这样，我们的文件系统之中便包含了一个可用的用户应用程序，接下来我们就来运行它。

代码 8-2

```
int exec(unsigned int start){
    asm volatile(
        "mov pc,r0\n\t"
    );
    return 0;
}
```

想要在操作系统的源代码中调用一个外部应用程序，首先应该有一个能够运行外部应用程序的函数，代码 8-2 就是这样一个函数。

前面我们已经提到过，在 ARM 体系结构中运行外部应用程序，mov 指令是一个不错的选择。使用这条指令将程序计数器 PC 的值赋值为外部应用程序的入口地址。结合过程调用标准，函数的第一个参数会保存到寄存器 R0 中。所以在代码 8-2 的 exec 函数里，程序简单地将 R0 的值赋值给 PC，目的是去运行地址 R0 处的代码。而函数 exec 的使用方法就是把外部的用户应用程序在内存中的地址作为参数传递给 exec 函数。

现在，请将代码 8-2 的内容保存成文件，名为“exec.c”。然后，修改 Makefile 中的 OBJS 变量来编译这个文件。接下来，我们还需要将文件“boot.c”稍做修改，其内容如下：

代码 8-3

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
```

```
    start_mmu ( ) ;
//  timer_init ( ) ;

    init_page_map ( ) ;
    kmalloc_init ( ) ;
    ramdisk_driver_init ( ) ;
    romfs_init ( ) ;

    struct inode *node;
    char *buf=(char *)0x30100000;

    if((node=fs_type[ROMFS]->namei(fs_type[ROMFS],\
                "main.bin"))==(void *)0){
        printk("inode read eror\n");
        goto HALT;
    }

    if(fs_type[ROMFS]->device->dout(fs_type[ROMFS]->device,\
                    buf,fs_type[ROMFS]->get_daddr(node),\
                    node->dsize)){
        printk("dout error\n");
        goto HALT;
    }

    exec(buf);
HALT:
    while(1);
}
```

在代码 8-3 中，程序首先通过 romfs 文件系统类型的 namei 函数得到一个 struct inode 结构体，并赋值给 node 指针，然后再调用该文件系统类型所寄生的存储设备的 dout 方法，将该 inode 表示的文件的实际内容读到内存 buf 处。

这里我们使用了一个非常不规范，甚至是危险的做法，将 buf 的指针强行指向 0x30100000 内存处，因为这段代码只是示例，同时，操作系统中目前并没有使用 0x03010000 这段内存空间，所以此处姑且可以这样用，并不会产生错误的结果。

那为什么一定要将 buf 指向 0x30100000 呢，别的内存地址不可以吗？别忘了，用户的外部应用程序是从 0x30100000 处开始运行的，这在程序编

译的时候就已经确定了。因此，程序只有被加载到该处才能够正常运行。

现在，请编译并运行程序，不出意外的话，程序的运行结果将会如下：

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
Loaded RAM   ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
this is a test application
```

可以看出，“this is a test application”字符串被成功打印出来了，这表明用户应用程序运行得正确无误。

上面我们实现了一个在自己的操作系统中调用外部程序的简单方法。就好像是在 Windows 系统中安装了一个应用程序，并成功运行了它。从此以后，凡是功能上的需求，都可以交给用户应用程序去完成。虽然我们的实现方法很简单，但至少给应用程序的正确运行铺平了道路。接下来我们就来慢慢地完善这段代码，让应用程序的调用过程更加正规和不受限制。

8.2 可执行文件格式

在前面的例子中，用户应用程序已经可以正常运行了。但是程序的运行却受到了限制，其中一点就是，操作系统必须将应用程序加载到指定的位置去运行。这样一来，如何能够更方便地知道一个应用程序的运行地址而不是简单地让开发者去约定，就成了一个必须要解决的问题。

解决这个问题的思路是，由应用程序根据操作系统的指导去选择自己的运行地址，然后应用程序想办法将该地址通知操作系统，操作系统利用各种手段构建出能够运行应用程序的环境，整个过程都是自动实现的，不需要程序员人为参与。

其中，应用程序选择运行地址和操作系统运行应用程序这两个步骤都没有

什么好说的，重点是应用程序如何将自身运行的必要信息通知给操作系统。

其实，让一个可执行的用户程序不仅仅包含代码和数据，还包含一些额外的信息即可。操作系统在运行程序时，首先读取并分析这些信息，检查这些信息的正确性，然后根据这些信息构建运行环境，还要将真正的代码从文件中提取出来，放到正确的位置上再去运行。

在这些信息中，最重要的一条就是程序的运行地址。其他信息还包括程序的版本，程序是使用什么样的编译器编译的，程序是运行在哪种硬件平台、哪个操作系统下的，等等。这些其他信息，有的时候能够避免应用程序由于版本不匹配、格式不正确或系统不兼容等问题而造成的运行错误。

既然应用程序包含了真正的代码和相关信息，那么这些信息采用什么样的格式进行存储就是我们需要考虑的问题了。其实格式并不重要，我们完全可以自定义一种格式，然后让我们的操作系统去支持它。当然，这是不明智的。对于可执行文件格式，更好的做法是选择一种主流标准并去实现它，于是我们选择了 ELF。

8.2.1　ELF 格式的组成结构

ELF 指的是 Executable and Linkable Format。最初是由 UNIX 系统实验室作为应用程序二进制接口开发和发行的，后来逐渐发展成为了可执行文件的格式标准，在很多操作系统和非操作系统环境中都有非常广泛的应用。完整的 ELF 格式标准涉及了三个方面的内容。在这里我们只需要关心一个方面，那就是一个 ELF 格式可执行程序的组成结构。

一个 ELF 可执行文件格式如图 8-1 所示。

像图 8-1 那样，一个 ELF 可执行文件包含了一个描述全局信息的 ELF 文件头、若干个 Program 头、若干个 Segment 以及若干个可有可无的 Section 头。在 Segment 中保存的正是程序的运行代码，而 Program 头描述了各个 Segment 和其他必要信息，如链接库信息、文件辅助信息等。在代码真正运行时，我们往往只关心实际的代码部分，而与运行无关的其他信息则可以忽略掉。

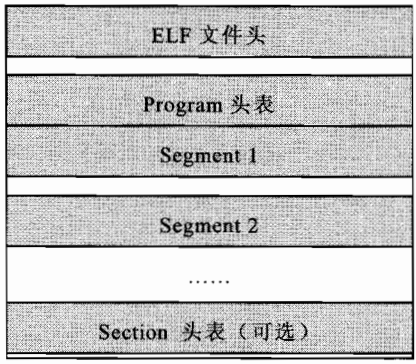

图 8-1　ELF 文件结构

当操作系统需要执行一个 ELF 格式的文件时，只需要分析 ELF 文件头，然后找到 Program 头的位置，依次分析这些 Program 头，找到代表代码和数据的 Segment，最后将这些 Segment 复制到指定的地址处，便可以运行程序了。

一个 ELF 文件头由如下部分组成：

代码 8-4

```
typedef unsigned int elf32_addr;
typedef unsigned int elf32_word;
typedef signed int elf32_sword;
typedef unsigned short elf32_half;
typedef unsigned int elf32_off;

struct elf32_ehdr{
    unsigned char e_ident[16];
    elf32_half  e_type;
    elf32_half  e_machine;
    elf32_word  e_version;
    elf32_addr  e_entry;
    elf32_off   e_phoff;
    elf32_off   e_shoff;
    elf32_word  e_flags;
    elf32_half  e_ehsize;
    elf32_half  e_phentsize;
    elf32_half  e_phnum;
    elf32_half  e_shentsize;
    elf32_half  e_shnum;
```

```
    elf32_half  e_shstrndx;
};
```

因为 ELF 可执行程序格式可以支持不同位数的处理器，所以 ELF 标准中自定义了一些专有的数据类型。无论是 8 位的处理器还是 32 位的处理器，这些数据类型的大小都是一致的，从而保证了文件与处理器格式的无关性。这些数据类型的大小和含义如表 8-1 所示。

表 8-1　ELF 格式中的数据规定

数据类型	大小	对齐	含义
elf32_addr	4	4	用于描述程序运行地址
elf32_word	4	4	描述无符号大整数
elf32_sword	4	4	描述有符号大整数
elf32_half	2	2	描述无符号中等大小的整数
elf32_off	4	4	描述无符号的文件偏移量

让我们结合表 8-1 的描述，逐个分析一下 ELF 文件中各部分的含义。

一个 ELF 文件头总是出现在文件的最开始处。其中，第一个成员是一个 16 个字节数组，里边记录了文件的标识、版本、编码格式等信息。

之后的两个字节是 e_type 成员，记录了目标文件属于 ELF 格式标准下的哪种类型，比如是可执行的文件还是可重定位文件，等等。根据 ELF 格式标准，这个值是 2，代表了这个文件是一个可执行的文件，这也正是我们需要的。

接下来的两个字节 e_machine 描述的是该程序运行的硬件平台。在我们的例子中，这个值必须是 40，表示这个应用程序是在 ARM 中运行的。

然后，4 个字节的空间 e_version 用于描述应用程序的版本，通常可以是 1。

接下来的 4 个字节就相当重要了，结构体成员 e_entry 记录了程序运行时的入口地址，也就是说，应用程序的第一条指令就应该出现在这个地址处。

e_phoff 成员记录了第一个 Program 头在文件内的偏移。e_phentsize 成员则代表了每一个 Program 头大小，再结合能够描述文件中共有多少个 Program 头的 e_phnum 成员，我们就可以遍历每一个 Program 头，并可以从中找到代码和数据在文件中的位置。

代码 8-4 中的其他成员与程序的执行关系不大，这里就不多介绍了。读者朋友们如果还对这些内容感兴趣，可以去查阅相关文档。

另外，还有一个问题需要解决。在遍历每一个 Program 头时，如何才能知道这个头信息所描述的 Segment 就是数据或代码，而不是与运行程序无关的其他信息呢？我们在 Program 头结构体中可以找到答案。

代码 8-5

```
struct elf32_phdr{
    elf32_word  p_type;
    elf32_off   p_offset;
    elf32_addr  p_vaddr;
    elf32_addr  p_paddr;
    elf32_word  p_filesz;
    elf32_word  p_memsz;
    elf32_word  p_flags;
    elf32_word  p_align;
};
```

代码 8-5 定义了 elf32_phdr 结构体用于描述 Program 头信息。在该结构体中，与段类型直接相关的是 p_type 成员，只有当该成员的值为 1 时，才表示该 Segment 是要运行的代码或数据，需要在执行时加载到内存中去。

一旦确定需要进行内存加载，之后要做的就是读取 p_offset 成员的值，它表示要加载到内存中的这个 Segment 在文件中的偏移量，再结合描述 Segment 大小的成员 p_filesz，就可以精确地定位程序的代码和数据了。

最后，程序还要读取 p_vaddr 成员，它表示这个 Segment 应该出现在内存的哪个位置。这样就可以将该 Segment 从文件中复制到正确的内存地址处了。

8.2.2 操作 ELF 格式文件的方法

综合以上的描述，总结执行 ELF 格式文件的方法，步骤如下：

（1）从文件起始位置读取一个 struct elf32_ehdr 结构体，验证文件的正确性以及文件与操作系统是否匹配。

（2）找到该结构体中 e_entry 成员，从系统中获得这个值所指向的内存地址。

（3）读出 struct elf32_ehdr 结构体中的 e_phoff、e_phextsize 以及 e_phnum 三个成员。根据这三个值，利用 struct elf32_phdr 结构体遍历文件中每一个

Program 头。

（4）在遍历的过程中，检查 struct elf32_phdr 结构体中的 p_type 成员，如果为 1，则调用存储设备的相关函数，将文件内偏移为 p_offset、大小为 p_filesz 的一段数据从存储器中读取到 p_vaddr 所指向的内存位置。

（5）调用执行函数，使程序从 e_entry 内存处开始执行。

这样，一个 ELF 格式文件的执行过程就可以顺利完成了，将上述步骤用程序来实现，如下：

代码 8-6

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
//  timer_init();
    init_page_map();
    kmalloc_init();
    ramdisk_driver_init();
    romfs_init();

    struct inode *node;
    struct elf32_phdr *phdr;
    struct elf32_ehdr *ehdr;
    int phnum,pos,dpos;
    char *buf;

    if((buf=kmalloc(1024))==(void *)0){
        printk("get free pages error\n");
        goto HALT;
    }

    if((node=fs_type[ROMFS]->namei(fs_type[ROMFS],"main"\
    ))==(void *)0){
        printk("inode read eror\n");
        goto HALT;
    }

    if(fs_type[ROMFS]->device->dout(fs_type[ROMFS]->device,buf,\
```

```
            fs_type[ROMFS]->get_daddr(node),node->dsize)){
        printk("dout error\n");
        goto HALT;
    }

    ehdr=(struct elf32_ehdr *)buf;
    phdr=(struct elf32_phdr *)((char *)buf+ehdr->e_phoff);

    for(i=0;i<ehdr->e_phnum;i++){
        if(CHECK_PT_TYPE_LOAD(phdr)){
          if(fs_type[ROMFS]->device->dout(fs_type[ROMFS]->device,\
                        (char *)phdr->p_vaddr,\
                        fs_type[ROMFS]->get_daddr(node)+\
                        phdr->p_offset,phdr->p_filesz)<0){
            printk("dout error\n");
            goto HALT;
          }
        }
        phdr++;
    }

    exec(ehdr->e_entry);
HALT:
    while(1);
}
```

在代码 8-6 中，程序首先通过 romfs 文件系统的 namei 函数读取 RAM 盘上的 main 文件，得到代表该文件的 inode 结构体。不同于代码 8-3 中的“main.bin”文件，这里的 main 文件是直接由编译器编译生成的 ELF 格式文件。于是，我们可以依据执行 ELF 程序的一般步骤对它进行处理。

首先要做的就是通过存储设备的 dout 函数从文件系统中读出 main 文件的内容，并保存到 buf 中。为了保证程序简单直观，这里我们没有验证缓冲区的大小是否足够装下 ELF 和 Program 头信息，程序仅仅通过 kmalloc 函数申请了一个 1K 大小的内存来存储文件开头的部分。这样做，运行本书的例子至少是没有问题的。

接下来，我们需要找到描述 ELF 头信息的结构体 struct elf32_ehdr 的位置，并通过读取它的 e_phoff 成员找到第一个 struct elf32_phdr 结构体。这两个结构体的地址，分别被保存在变量 ehdr 和 phdr 中。

然后，程序从 phdr 变量开始，通过循环读取每一个 Program 头信息，依次判断 struct elf32_phdr 结构体的 p_type 成员是否为 1。如果该成员为 1，则表示此 Program 头所描述的正是代码段或数据段。此时需要再次调用 dout 函数，将用户应用程序的代码或数据从存储设备中复制到内存里。这两个信息分别记录在了 struct elf32_phdr 结构体的 p_offset 和 p_vaddr 结构体的成员中。

当所有的代码和数据最终都被加载到内存的正确位置后，就可以调用 exec 来运行用户应用程序了。用户程序的入口地址可从 struct elf32_ehdr 的 e_entry 成员中得到。

有了这样的方法，操作系统在运行用户程序时，便不再需要了解用户程序的细节。而仅需从应用程序中读出与程序运行有关的信息，根据这些信息的提示准备好程序运行的环境。这样，理论上就实现了运行任意二进制应用程序的可能。

然而从另一个角度来看，这种运行应用程序的方法还存在一个致命的缺陷，那就是我们无法保证用户应用程序的运行地址恰好是有效的。

这需要分两种情况去分析。第一种情况是，用户应用程序的运行地址已经超出了物理内存的范围，例如，一个 ELF 格式的用户应用程序需要运行在内存为 0x40000000 的地址中，但我们的虚拟系统，不计算 RAM 盘所占用的内存空间，一共只有 8M 的可用内存，0x40000000 这个地址远远超过了硬件原有内存的范围。第二种情况是，用户应用程序的运行地址虽然落在了物理内存的范围内，但这个地址却恰好被别的应用程序提前占用了。无论哪一种情况发生，都会使我们的操作系统将一个原本可以正常运行的用户应用程序拒之门外。

想要解决这个问题，可以通过虚拟内存映射将原本无效的内存地址映射到有效的空间中。于是，我们只需要将应用程序的代码和数据保存到任意一个没有被使用的有效内存中，然后修改页表，将这个内存地址映射到用户应用程序规定的地址。由此，读者也可以深入地体会一下虚拟地址映射对于一个功能强大的操作系统来说是多么的重要。

当然，想要实现这样的功能，我们的操作系统代码需要进行很大的调整，这其中至少包含内存管理部分和 MMU 部分两个子结构。为了不占用过多的篇幅，这段代码我们就不去具体实现了。读者如果感兴趣的话，可以尝试自

行实现。

下面我们来实践一下这段代码。

8.2.3 运行 ELF 格式的应用程序

首先我们需要提供一些与 ELF 格式有关的宏定义。

代码 8-7

```
#define ELFCLASSNONE  0
#define ELFCLASS32    1
#define ELFCLASS64    2
#define CHECK_ELF_CLASS(p)                ((p)->e_ident[4])
#define CHECK_ELF_CLASS_ELFCLASS32(p) \
(CHECK_ELF_CLASS(p)==ELFCLASS32)

/*definition of elf data*/
#define ELFDATANONE       0
#define ELFDATA2LSB       1
#define ELFDATA2MSB       2
#define CHECK_ELF_DATA(p)                 ((p)->e_ident[5])
#define CHECK_ELF_DATA_LSB(p) \
    (CHECK_ELF_DATA(p)==ELFDATA2LSB)

/*elf type*/
#define ET_NONE           0
#define ET_REL        1
#define ET_EXEC           2
#define ET_DYN        3
#define ET_CORE           4
#define ET_LOPROC     0xff00
#define ET_HIPROC     0xffff
#define CHECK_ELF_TYPE(p)             ((p)->e_type)
#define CHECK_ELF_TYPE_EXEC(p) \
   (CHECK_ELF_TYPE(p)==ET_EXEC)

/*elf machine*/
#define EM_NONE       0
#define EM_M32        1
#define EM_SPARC      2
#define EM_386        3
```

```
#define EM_68k          4
#define EM_88k          5
#define EM_860          7
#define EM_MIPS             8
#define EM_ARM          40
#define CHECK_ELF_MACHINE(p)          ((p)->e_machine)
#define CHECK_ELF_MACHINE_ARM(p)\
    (CHECK_ELF_MACHINE(p)==EM_ARM)

/*elf version*/
#define EV_NONE             0
#define EV_CURRENT      1
#define CHECK_ELF_VERSION(p)                ((p)->e_ident[6])
#define CHECK_ELF_VERSION_CURRENT(p)\
    (CHECK_ELF_VERSION(p)==EV_CURRENT)

#define ELF_FILE_CHECK(hdr) ((((hdr)->e_ident[0])==0x7f)&&\
                             (((hdr)->e_ident[1])=='E')&&\
                             (((hdr)->e_ident[2])=='L')&&\
                                 (((hdr)->e_ident[3])=='F'))

#define PT_NULL                 0
#define PT_LOAD                 1
#define PT_DYNAMIC              2
#define PT_INTERP               3
#define PT_NOTE                 4
#define PT_SHLIB                5
#define PT_PHDR                 6
#define PT_LOPROC               0x70000000
#define PT_HIPROC               0x7fffffff
#define CHECK_PT_TYPE(p)            ((p)->p_type)
#define CHECK_PT_TYPE_LOAD(p)     (CHECK_PT_TYPE(p)\
        ==PT_LOAD)
```

代码 8-7 中，有一部分内容我们曾经使用过，但大部分宏在本书的代码中都未曾用到。这些宏都是根据 ELF 文件格式标准定义的，具有一定的通用性。朋友们也可以将这些宏用到自己的代码之中。

现在，读者朋友们可以将代码 8-7 与代码 8-5、代码 8-4 的内容共同保存到一个文件之中，命名为“elf.h”。

然后修改“boot.c”文件，先在文件的开始处将“elf.h”文件包含进来，

再把 plat_boot 函数修改为代码 8-6 那样。

这样，操作系统部分的代码就算完成了，读者可以尝试编译一下程序，确保没有语法错误。

接下来，我们还需要重新制作一个 RAM 盘存储设备映像。进入 tools 目录，将编译生成的测试应用程序 main 复制到 tools 中的 filesystem 目录。使用工具制作一个新的 romfs 文件系统类型文件，并将这个文件复制到上级目录的“ram.img”文件中。

现在回到我们的操作系统根目录，启动虚拟机，运行操作系统。不出意外的话，您将看到与上一节相同的结果。

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
Loaded RAM   ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
this is a test application
```

8.3 系统调用

现在，我们的操作系统已经拥有了运行和管理用户应用程序的能力，那些五花八门的功能和产品需求都可以交给用户应用程序去完成。此刻，我们的操作系统距离最终的无所不能，只差最后一步了，而这一步就是系统调用。

8.3.1 用户和内核的运行空间

想要讲清楚系统调用的来龙去脉，我们需要首先来说一说另外一对概念，那就是用户空间与内核空间。

这里的空间，可以简单地理解成运行环境。所以，我们可以简单地将用

户空间解释成为用户应用程序的运行环境，而内核空间则代表了操作系统所运行的环境。

通常对于单内核结构的操作系统来说，操作系统自身将会运行在内核空间，它拥有对整个硬件平台的绝对控制权。因此，当一段程序运行在内核空间中，既可以说明该程序应属于操作系统内核的一部分，又说明这段程序可以毫无限制地对计算机为所欲为。

与之相对的就是普通应用程序了。操作系统的作用是为应用程序提供必要的服务和支持，同时也会进行适当的监督和控制。而应用程序的行为对于操作系统来说是未知的，也就是说操作系统并不知道应用程序将做什么事。这样，当一个应用程序企图破坏整个系统的正常秩序、恶意操纵和控制其他应用程序时，就会产生极其严重的后果。

引入用户空间这种思想正是为了解决这一问题。一个用户应用程序从诞生那天开始就只能工作在用户空间，而处在用户空间的程序只有在自己的一亩三分地里才可以作主，决不允许多管别人的闲事。

这样，操作系统将自己运行在内核空间，而将应用程序限制在用户空间。同时，操作系统还将 CPU、内存、外设等几乎所有的资源都划归自己直接去管辖，不给用户空间直接访问的权力。此时，一个普通的用户应用程序即使有天大的本领，也不可能惹出什么乱子来，从而保证了各应用程序之间公平有序地运行，更好地去实现其功能。

但如此一来，当某个应用程序确实需要合理地使用系统中某些资源时，就会变得很困难。

如果这种情况真的发生，应用程序就不得不首先向操作系统做出申请。操作系统在确定应用程序是出于正当目的后，就会尽最大努力满足用户的要求，也只有这样，才能实现系统安全和运行效率之间的平衡。

于是，我们可以将用户应用程序向操作系统提出申请的这一动作叫做系统调用。

系统调用的实施过程可以通过图 8-2 进行描述。

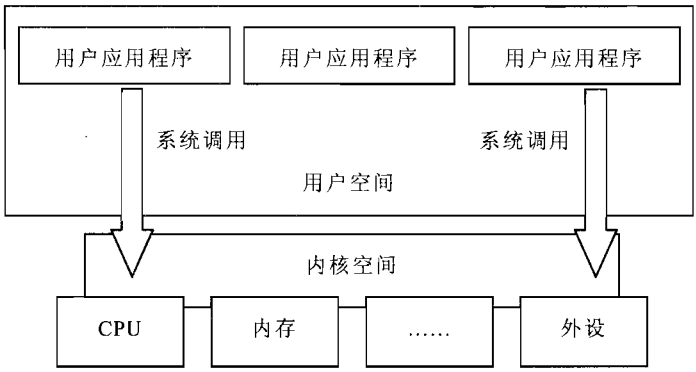

图 8-2　系统调用示意图

这里我们还需补充的是，图 8-2 并不能代表所有高端操作系统的系统调用结构。有些操作系统，如 Windows，甚至连内核自身也不具备控制所有资源的能力。可以说，图 8-2 的描述只适用于单一内核的高端操作系统。

有的读者可能会思考这样一个问题，操作系统怎样能够保证运行于用户空间的代码就一定不具备访问系统资源的能力呢？或者可以这样说，当某个应用程序硬是要强行访问系统或别的应用程序的资源，我们拦得住吗？

这个问题的答案是，拦不住。无论操作系统程序想什么样的办法，都没有能力彻底限制用户程序去做它想做的事。因为从 CPU 的角度上看，操作系统程序与用户应用程序同为代码，本没有任何区别。

正是因为只有 CPU 才有权力去限制程序的运行，所以这种用户空间和内核空间的划分，本质上讲是由 CPU 来完成的。CPU 只是让操作系统默认出生在了具有特权的内核空间，然后授权操作系统，让操作系统去管理用户程序，使用户程序生长在没有特权的用户空间，这样就自然而然地限制了运行在用户空间的应用程序。

回忆一下 ARM 体系结构的 7 种模式。在这 7 种模式中，用户模式与其他 6 种模式相比，不具备访问系统资源的能力，如全局中断控制等。因此，用户模式就成为了限制应用程序越权的天然屏障。再比如 x86 系列 CPU，保护模式下特权级拥有 4 个 Level，其中运行在 Level 0 中的代码享有至高无上的权力，适合作为内核空间运行操作系统程序，而运行于 Level 3 中的代码，其能力却极为有限，适合作为用户空间来运行用户应用程序。

既然用户模式总是工作在低特权的环境中，那么一旦一个应用程序确实需要使用系统资源时，就不得不想办法从低特权模式跳跃到高特权模式，然后才能去实施。这也成为了系统调用的另外一个特点。

因为低特权级的程序没有办法直接调用高特权级的程序，所以系统调用的实现不能简单依靠函数间调用，而必须使用别的方法。在多数体系结构中，操作系统都是通过软件中断来实现系统调用的。

8.3.2　实现一个系统调用

接下来，我们不如也来尝试实现一个系统调用接口。一旦系统调用能够轻松实现，那么在用户空间封装标准库函数就不再困难，一个结构完整的操作系统也将指日可待。无论从哪个角度看，这都是非常有意义的。

8.3.2.1　软中断的使用

为了实现一个系统调用，我们需要给目前的操作系统内核动手术。本质上讲，系统调用的内核实现其实就是软中断的处理问题。想要使系统调用得以运行，首先就要激活软中断。于是，我们首先要做的就是修改与异常向量有关的代码。一个最简单的软中断向量表的形式如下：

代码 8-8

```
__vector_SWI:
  stmfd r13!,{r0-r3,r14}
  ……
  <handler code>
  ……
  ldmfd r13!,{r0-r3,pc}^
```

代码 8-8 是软中断的一个简单结构。当软中断发生时，经过异常向量表的索引，程序会跳转到标签__vector_SWI 处运行。在此处程序首先将参数寄存器 R0~R3 的值保存到堆栈之中，同时将上一个状态的返回值也压入堆栈。

回忆一下前面我们讲过的内容，不同异常发生时，返回值的修正方法是不同的。对于软中断来说，寄存器 R14 保存的恰是上一个状态的返回值，因此不需要修正。当软中断的代码操作完成后，程序依次将堆栈中的值恢复到寄存器中，返回到原状态继续运行。

这段软中断的实现方法虽然简单，但是用在一个功能强大的操作系统当中却并不合适，原因主要有下面两点。

第一，由于异常模式在切换时默认是关闭中断的。因此当系统调用发生时，程序在内核中运行，将不能够被其他进程，甚至是中断处理程序所中断。这样一来，程序就失去了其实用价值。

第二，软中断默认工作在管理模式，而我们的操作系统却是运行在系统模式下的。这就意味着程序不得不首先切换到系统模式，然后才能执行用户请求的相关动作，这势必会影响系统调用的执行效率。

因此针对 ARM 体系结构，使用一些更高级的方法实现软中断对于系统调用来说更加有效。

下面这段代码给出了一种实现方法。

代码 8-9

```
__vector_SWI:
    str r14,[r13,#-0xc]
    mrs r14,spsr
    str r14,[r13,#-0x8]
    str r0,[r13,#-0x4]
    mov r0,r13
    CHANGE_TO_SYS
    str r14,[r13,#-8]!
    ldr r14,[r0,#-0xc]
    str r14,[r13,#4]
    ldr r14,[r0,#-0x8]
    ldr r0,[r0,#-0x4]
    stmfd r13!,{r0-r3,r14}
    ldr r3,[r13,#24]
    ldr r0,[r3,#-4]
    bic r0,r0,#0xff000000
    ldr r1,[r13,#32]
    ldr r2,[r13,#36]
    bl  sys_call_schedule
    str r0,[r13,#28]
    ldmfd r13!,{r0-r3}
    ldmfd r13!,{r14}
    msr cpsr,r14
    ldmfd r13!,{r14,pc}
```

在代码 8-9 中，一开始程序运行在管理模式，为了能够正确处理软中断，

程序必须想办法在第一时间切换到系统模式中去。因此，我们需要将用户模式下的状态和返回地址保存到管理模式下的堆栈中。

之后我们还需要将管理模式下的堆栈指针保存到 R0 寄存器中，以便在系统模式下也可以访问到管理模式堆栈中的内容。因为一旦程序切换到系统模式后，管理模式的堆栈指针寄存器 R13 将不可访问。

当一切准备工作都完成后，程序通过一个叫做 CHANGE_TO_SYS 的宏切换到系统模式，该宏的实现方法如下：

代码 8-10

```
.macro CHANGE_TO_SYS
   msr   cpsr_c,#(DISABLE_FIQ|DISABLE_IRQ|SYS_MOD)
.endm
```

一旦程序切换到系统模式，首先需要做的就是保存系统模式下的 R14 寄存器，以防止子程序调用时子程序的返回值将原来存储在 R14 寄存器中的值改写。

紧接着，程序利用 R0 寄存器依次读出用户模式的返回值和状态信息，它们也将保存到系统模式中。这样一来，程序就可以从系统模式中直接返回到调用之前的状态，而不需要再度切换回管理模式了。

于是这一段代码运行完之后，系统模式下的堆栈数据就如图 8-3（a）所示（图 8-3（b）为完成 8.3.2.2 节的过程后的程序堆栈）。

代码 8-9 接下来的工作就比较关键了。程序先将当前堆栈向高地址偏移 24 个字节处的数据读到寄存器 R3 中。然后再以这个值为地址，取该地址前面的 4 个字节的数据保存到 R0 中。读者可以再看一下图 8-3（a）。系统模式下的 R13 向高地址偏移 24 个字节，此处存储的正是用户模式的返回地址。结合前面章节的描述，这用户模式的返回值，记录的不正好是软中断指令的下一条指令地址吗？那么如果将这个地址向低地址处偏移的 4 个字节内容读出来，那不正好就是软中断指令自身吗？于是，上述动作完成后，R0 中保存的就是软中断指令的机器码。

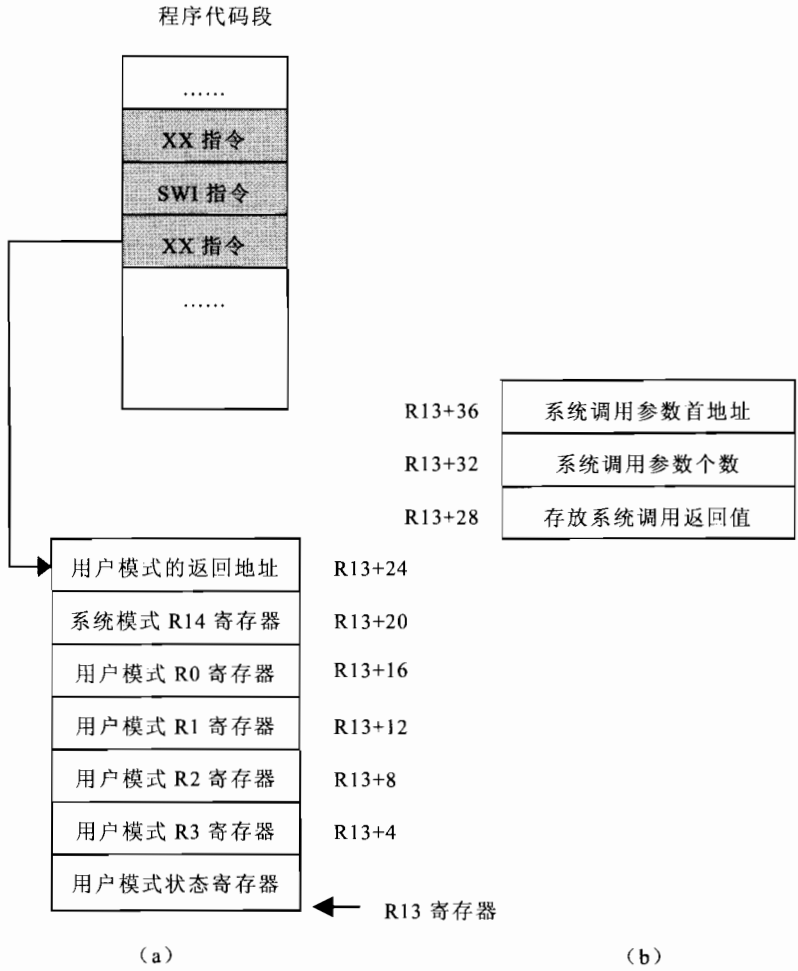

图 8-3　系统调用过程中的堆栈设计

但问题接着就来了，我们费这么大劲儿得到软中断的机器码又有什么意义呢？

这就需要从 ARM 体系结构中的软中断指令结构说起了。

众所周知，ARM 指令集中的指令都是 32 位等宽的，软中断指令自然也不例外。ARM 的软中断指令 SWI 由两部分组成，其机器码格式为 0xEFXXXXXX。由此可知，SWI 这条指令的高 8 位为 0xEF，其值是固定的，低 24 位则可以是任意数值，不会影响该指令的运行。而这个 24 位数，通常会被当成是软中断的参数在软中断发生时传递给处理程序。

因此，我们可以利用前面所描述的方法将这 24 位数值读出来，然后提取出软中断调用参数，并根据这个参数去完成不同的功能。在系统调用过程中，这个参数也就直接或间接地成为了标识系统调用号的重要手段。

系统调用号就是系统调用的代号，绝大多数高级操作系统都不只有一个系统调用。这么多的系统调用应该如何管理？很显然数字编号是最简单的一种方式。同时，由于多数操作系统会在系统调用的基础之上再封装一些函数接口供用户使用，这样一来，系统调用号所带来的使用上的不便也就不明显了。

现在回到程序当中。当系统能够正确得到系统调用号之后，程序又依次将 R13 寄存器向高地址偏移 32 个字节和 36 个字节的这两个数据从内存中加载到 R1 和 R2 两个寄存器中。

之后，程序调用了一个叫做 sys_call_schedule 的函数。毫无疑问，这里的 R1、R2 和前面的 R0 寄存器其实都是传递给 sys_call_schedule 函数的参数。其中，该函数的第一个参数就是系统调用号，而第二个和第三个参数是预先被存储到内存中的，我们稍后会解释。

sys_call_schedule 函数返回后，程序将寄存器 R0 的值保存到了 R13 向高地址偏移 28 个字节处。根据过程调用标准，函数返回后会将返回值保存到寄存器 R0 中。这也就说明保存到这段内存中的数据，恰好就是 sys_call_schedule 函数的返回值了。

最后，当寄存器 R0~R3、状态寄存器、系统模式下的 R14 寄存器以及用户模式的返回地址依次从堆栈中弹出时，系统调用过程结束，程序返回到用户空间当中。

8.3.2.2　为用户程序提供系统调用接口

系统调用的具体实现都是在内核空间中完成的，那么用户空间的应用程序又该做哪些工作才能成功触发某个系统调用呢？

正是通过代码 8-11 中 SYSCALL 这个宏，用户应用程序的系统调用申请才得以实现，下面我们就来简要分析一下这个宏的实现方法。

代码 8-11

```
#define __NR_SYSCALL_BASE   0x0
```

```
#define  __NR_test              (__NR_SYSCALL_BASE+ 0)
#define  __NR_SYS_CALL      (__NR_SYSCALL_BASE+1)

typedef int (*syscall_fn)(int num,int *args);

#define SYSCALL(num,pnum,parray,ret) do{       \
                              asm volatile(   \
                                "stmfd r13!,{%3}\n"\
                                "stmfd r13!,{%2}\n"\
                                "sub r13,r13,#4\n"\
                                "SWI %1\n"\
                                "ldmfd r13!,{%0}\n"\
                                "add r13,r13,#8\n"\
                                :"=r"(ret)\
                                :"i"(num),"r"(pnum),"r"(parray)\
                                :"r2","r3"\
                                );\
                            }while(0)
```

我们先来聊一聊这个宏的实现结构。有的读者可能会对这种 do{}while(0)的结构非常好奇，其实，这是一种保证宏定义能够在各种应用条件下都能成功编译的有效方法。假如，我们想定义一个由多个代码块组成的宏 MACRO，其形式如下：

```
#define MACRO  first_step;second_step;third_step
```

那么如果某个程序员是以下面这样的方式使用这个宏，就会造成编译错误。

```
if(condition)
    MACRO;
else
    do_something_else;
```

很显然，我们不可能强制要求用户在使用 MACRO 宏时必须在 if 结构中加上{}。所以，为了尽可能减少类似问题造成的编译错误，就需要在定义宏的时候使用一个看似无用的 do{}while(0)结构了。在这个例子中，如果宏 MACRO 被定义成这样：

```
#define MACRO   do{first_step;second_step;third_step;}while(0)
```

那么经过预处理后，程序将展开成如下形式：

```
if ( condition )
    do{first_step;second_step;third_step;}while ( 0 ) ;
else
    do_something_else;
```

这样一来，do{}while（0）中包含的代码片段就会被编译器认为是一个完整结构，从而保证了程序的正确编译。

SYSCALL 宏最终是通过 C 语言内嵌汇编的方法实现了系统调用的请求的。有关在 GNU 环境下 C 语言内嵌汇编的具体方法，我们在前面的章节已经介绍过了，如果读者忘记了，可以翻看前面的内容。虽然 SYSCALL 宏是用汇编语言实现的，但却不难理解。程序是按照“压入堆栈，预留空间，触发系统调用，提取返回值，修正堆栈指针”的顺序完成系统调用申请的。

在整个过程中，最重要的应该是 SWI 指令。为了保证能够使用同一个系统调用接口调用操作系统内核中不同的函数，程序在使用 SWI 指令的同时，又将系统调用号传递到内核中去作为操作系统内核中不同函数的一个索引。

但即使这样，系统调用仍不足以发挥它的威力。举一个 UNIX 中最常用的例子，在类 UNIX 操作系统中打开一个文件是通过系统调用 open 来实现的。在 minix 下，open 的系统调用对应的号码为 5。我们可以假定，用户最终会通过类似于 SWI 5 这样的指令产生一个软中断，并且在内核中利用 5 这个软中断号最终索引到某一个 open 函数，但因为 open 系统调用实现的是打开文件的功能，所以用户在产生系统调用时，除了要使用到系统调用号之外，还必须想办法将一些参数传递给内核，如要打开的文件名以及要以什么样的方式打开，等等。

那么用户空间的程序应该通过什么方法，才能在系统调用时将参数传递给内核空间呢？

也许读者朋友们首先想到的就是过程调用标准。按照标准中的规定，我们可以将系统调用的参数通过寄存器传递到内核空间。然而，这种做法将会带来一个问题，那就是当系统调用参数多于可用寄存器个数时，程序的执行就会遇到麻烦。

所以如果有一种方法能够适应任意个数的参数传递，那么程序将变得更加易用。读者可能首先想到的就是通过变参函数的方法来实现。

的确变参函数完全可以解决这个问题，但使用变参函数的实现方法，程序运行的开销却比较大，而系统调用又是一个频繁的动作。这样，变参函数的处理器开销会因此被放大，进而影响系统整体的运行效率。

所以这里我们选取了一个折中的方法，将所有需要传递给内核的参数都强制转换成32位整型数，保存到一个数组当中。当用户调用SYSCALL时，只需将数组首地址和数组成员的个数分别传递给内核，内核一次性从数组中读出参数并进行处理。不同于变参函数可以支持任意类型的特点，我们的方法只能使用整型数据，从而在保证了传递任意参数的同时也简化了参数的传递开销。

于是在代码8-11中，程序首先将参数数组的首地址parray压入堆栈。紧接着，将系统调用的参数个数pnum也压入堆栈。请注意，SYSCALL这个宏是内核专门给用户封装的，使用户能够方便地申请一个系统调用。因此，SYSCALL中所有的代码都将运行在用户空间。这就表示parray和pnum这两个值会被压入用户模式下的堆栈中。

事实的确如此，但是我们却不需担心，因为用户模式和系统模式共享同一个堆栈。这样，操作系统就可以无缝地访问用户程序的任何数据了。

再次回到代码当中，接下来程序通过将堆栈指针寄存器的值减1，在堆栈中预留了4个字节的内存空间，这段空间是专门用于保存内核系统调用实施函数的返回值的。

以上所有过程完成后，程序堆栈就变成了如图8-3（b）所示的那样了。

之后程序使用SWI指令产生一个软中断，然后将跳转到代码8-9的__vector_SWI处去运行。

当控制权再次从内核中移交到用户空间之后，程序将内核系统调用函数的返回值从堆栈中读出，赋值给ret，在对堆栈指针寄存器进行修正后，程序结束退出。

8.3.2.3 通用的系统调用函数

现在，代码8-11和代码8-9的功能都已经清楚了。系统调用也只剩下最后一个面纱，那就是代码8-9中的sys_scall_schedule函数。它代表了一个通用的系统调用函数。

代码 8-12

```
#include "syscall.h"

void test_syscall_args(int index,int *array){
    printk("this following message is from kernel printed by
test_syscall_args\n");
    int i;
    for(i=0;i<index;i++){
        printk("the %d arg is %x\n",i,array[i]);
    }
}

syscall_fn __syscall_test(int index,int *array){
    test_syscall_args(index,array);
    return 0;
}

syscall_fn syscall_table[__NR_SYS_CALL]={
    (syscall_fn)__syscall_test,
};

int sys_call_schedule(unsigned int index,int num,int *args){
    if(syscall_table[index]){
        return (syscall_table[index])(num,args);
    }
    return -1;
}
```

函数 sys_call_schedule 共有三个参数，分别是代表系统调用号的 index，代表系统调用参数个数的 num，以及参数数组的首地址 args。在代码 8-9 中调用该函数之前，这些参数已经被压入到堆栈里了。

于是，sys_call_schedule 函数首先要读取 syscall_table 数组的具体值，用来判断是否为空，若不为空，就表示在系统调用号 index 处已经注册了一个系统调用函数。然后执行这个函数，返回该函数的运行结果，如果 syscall_table 数组的 index 索引处的相应函数指针为 NULL，那就说明该系统调用并没有注册，于是返回-1。

每一个系统调用都是 syscall_fn 类型的函数。该类型定义在了代码 8-11 中，系统调用号和最大系统调用号也在该处定义。对于一个成熟的操作系统来说，一系列精心设计的系统调用函数是必不可少的。在这里我们仅想通过

一个小例子来说明问题，不会对系统调用函数的设计做深入的研究。

程序中我们只实现了一个调用号为 0 的系统调用，它所对应的函数原形为__syscall_test，这个函数仅仅实现了依次打印用户传递给内核的参数的功能以证明代码的正确性。

最后，程序还需要将所有的系统调用函数通过静态的方法保存到 syscall_table 数组中。

8.3.3 运行系统调用程序

一个系统调用的整个过程已经完成了，当然我们还需要亲自运行一下这段代码，才能更深入地理解系统调用的内涵。

首先，将代码 8-9 和代码 8-10 的内容添加到文件“abnormal.s”中，并将该文件中如下内容删除：

```
__vector_swi:
    nop
```

然后将代码 8-11 的内容保存成文件，命名为“syscall.h”。同时将代码 8-12 的内容保存成文件，取名“syscall.c”。

最后，别忘记修改 Makefile 文件中的 OBJS 变量，添加 syscall.o 目标文件。

至此，操作系统内核中的系统调用代码就顺利完成了。由于没有用户的触发，这样的代码目前尚不具备运行的条件。我们还需要写一个用户应用程序，并在用户程序中使用系统调用接口触发内核中的 0 号系统调用函数。程序的实现过程如下：

代码 8-13

```
#include "../syscall.h"

int main( ){
    const char *p="this is a test application\n";
    while(*p){
        *(volatile unsigned int *)0xd0000020=*p++;
    };

    int test_array[2],ret;
```

```
    test_array[0]=0xf0;
    test_array[1]=0x0f;

    SYSCALL( __NR_test,2,test_array,ret );
}
```

不同于前面几小节的用户应用程序，在代码 8-13 中，程序使用了操作系统内核为我们封装好的 SYSCALL 宏，这个宏能够通过软中断指令将系统从用户模式切换到系统模式中，进而执行内核中的系统调用函数。

__NR_test 宏展开后的值是 0，表示调用内核中 0 号系统函数。接下来传递的是该系统函数所需要的参数个数和参数列表。最后，程序还传入了一个未初始化的变量 ret，用来存储内核系统调用函数的返回值。

这样，用户空间的系统调用代码就完成了。需要注意的是，在这段用户应用程序中，SYSCALL 和__NR_test 等宏定义来源于操作系统源代码头文件“syscall.h”，我们可以在代码 8-13 中包含这个头文件，来保证用户使用的系统调用方法与内核代码中的定义方法一致。

现在请编译这段程序，并将生成的“ram.img”映像文件保存到我们的操作系统根目录中，然后切换到操作系统根目录下，编译操作系统代码，最后运行 skyeye 命令，如果一切顺利的话，您将会看到如下运行结果：

```
Your elf file is little endian.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
Loaded RAM   ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
this is a test application
this following message is from kernel printed by
test_syscall_args
the 0 arg is 0xf0
the 1 arg is 0xf
```

这表示系统调用被正确运行了。

8.4 总结

总的说来，本章解决的是如何在操作系统中自由地运行外部应用程序的问题。这个问题很重要，但是却经常被忽视。很多嵌入式操作系统不重视或限制用户应用程序自由地发挥。这样造成的结果就是大量的应用不得不由操作系统开发人员亲自操刀完成，从长远的角度来看，这样的系统没有生命力。

因此，一个更好的结构是让别人帮助你去完成系统的应用。就像Windows一样，操作系统的内核是由Microsoft开发的。但是，在Windows操作系统中的那些五花八门的应用程序则交给全世界的程序员去完成。同时，Microsoft也在尽最大努力，通过提供编译器、丰富的文档、程序接口等手段，保证这些程序在Windows操作系统中能够可靠运行。

另一个例子是Android。Android其实就相当于一个Linux针对手机的发行版。既然是与Linux有关的开发，并且系统又是运行在手机这种处理器性能相对较低的硬件平台中，使用C语言开发应用程序应该是理所当然的事。但是Android却大量提供了以Java为基础的运行环境，力挺Java作为上层应用的开发语言，这么做的目的之一，就是让尽可能多的程序员去使用一种相对简单的编程语言，更加容易地为Android编写应用程序。

把应用程序交给第三方开发人员去完成，操作系统的开发者就可以更加专注于系统自身的提升，而不会被各种各样的应用需求搞得焦头烂额。更重要的一点是，依存于某款操作系统进行应用程序开发的人越多，对这种操作系统的依赖程度就越大，操作系统本身就越来越不可替代。例如，在PC操作系统这个领域里，没有人能够取代Windows系列操作系统的地位，因为我们所熟悉的所有软件、操作方法、概念、规则标准等，统统都是基于Windows操作系统的。除非有一天PC不复存在了，否则Microsoft将会永远财源滚滚。要说这种运营战略有多高明倒不见得，这只是一种共赢的思想罢了，只不过鸡鸭鱼肉要进自己的肚子里，而那些剩菜剩饭，就分给应用程序的开发者好了。可悲的是，现在有相当多的一群人，还在为自己能够写出一些应用程序，赚点小钱而沾沾自喜呢！